Universidad Zaragoza

ITAINNOVA
INSTITUTO TECNOLÓGICO DE ARAGÓN

Trabajo de Fin de Grado de Física

Estudio de nuevos modelos de Deep Leaning para el análisis y comprensión de grandes cantidades de datos

---

Departamento de Ingeniería Electrónica y Comunicaciones
Departamento de Big Data y Sistemas Cognitivos - ITAINNOVA

Autor:
Miguel Lahoz Muñoz

Directores:
Rafael Del Hoyo Alonso
Nicolás Medrano Marqués

Septiembre 2019

Resumen

Las redes neuronales se están consolidando como método de resolución de problemas de difícil modelización: desde comportamiento predictivo, hasta modelos del lenguaje. En este trabajo se va a estudiar qué es una red neuronal y su modelización matemática, poniendo especial hincapié en las redes orientadas al tratamiento del lenguaje. Se explicarán los modelos Encoder-Decoder, junto a los módulos Transform, estudiando los más populares actualmente y comparando sus resultados en diversas tareas, finalizando con el estudio más completo de un modelo.

Índice de figuras

Índice de tablas

Índice general

Capítulo 1
Motivación y objetivos

En la actualidad, las redes neuronales se están utilizando para la resolución de una amplia variedad de problemas, desde la clasificación de datos [3] hasta la creación de obras musicales [18]. Uno de los campos en los que más se está desarrollando es en modelos de lenguaje, donde tenemos desde clasificación de textos, traducción de textos, hasta modelos responsivos del lenguaje. Gracias a los nuevos avances en Transofmers [19] y a nuevos algoritmos más eficientes [15] se están alcanzando nuevos hitos en las distintas pruebas usadas para medir los modelos de lenguaje.

En este trabajo esperamos comprender el funcionamiento matemático de estos nuevos modelos, estudiando además la eficiencia de estos algoritmos, y el tiempo necesario para obtener resultados decentes.

Capítulo 2
Introduccion

2.1. Enfoque biológico

Cuando tenemos un sistema que queremos estudiar, empezamos realizando modelos simples que nos permitan estudiar el comportamiento de dicho sistema con el que podemos trabajar. Cuando nos centramos en el estudio de las neuronas y las redes neuronales del cerebro, nos encontramos con diversos modelos [4]:

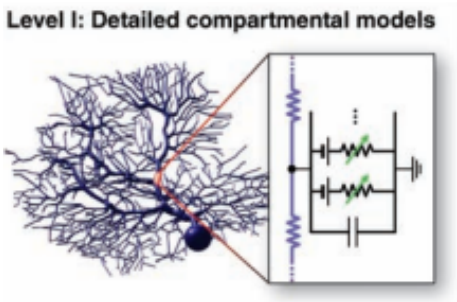

Figura 2.1: Modelo detallado de compartimentos **Fuente:** [4]

En la figura 2.1 tenemos el modelo detallado de compartimentos, un modelo que intenta aproximarse lo máximo posible al comportamiento real de una neurona mediante un circuito eléctrico equivalente. Este modelo está basado en datos experimentales del voltaje que aparece en una neurona en reposo y cuando transmite información. Reproduce fielmente el comportamiento de una neurona, pero debido a su complejidad no se puede utilizar en la simulación de redes más complejas.

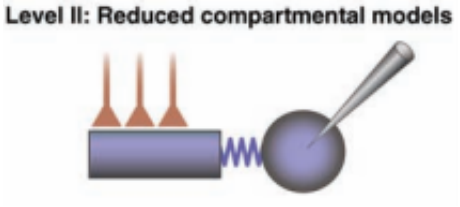

Figura 2.2: Modelo reducido de compartimentos **Fuente:** [4]

En la figura 2.2 aparece la primera simplificación del modelo, basada en un modelo de dos o tres compartimentos. Estos modelos permiten hacer simulaciones más complejas en menos tiempo, siendo a

la vez un modelo realista para estudiar ciertos comportamientos de las neuronas, como la importancia de las corrientes de calcio.

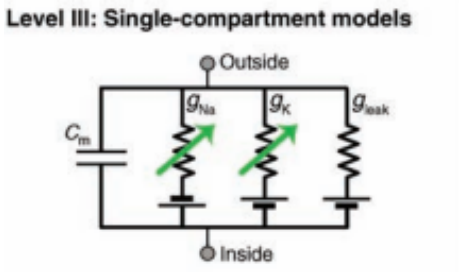

Figura 2.3: Modelo de un compartimento **Fuente:** [4]

Este modelo se puede simplificar aún más, llegando al modelo de un sólo compartimento que tenemos en la figura 2.3, también llamado modelo de Hodgkin-Huxley [6], el cual está basado en dos voltajes variables, g_{Na} y g_K, dependientes de la concentración de iones de Na^+ y K^+, un voltaje fijo de fuga g_{leak}, y una capacitancia que conecta la entrada y la salida C_m. Este modelo permite estudiar de forma muy eficiente los valores de entrada y salida de una neurona, dejando de lado lo que ocurre dentro de la misma.

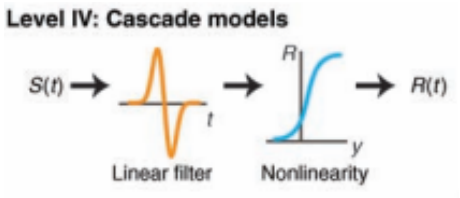

Figura 2.4: Modelos de cascada **Fuente:** [4]

En la figura 2.4 tenemos una aproximación más sistemática, en la que cada entrada de una neurona S(t) es filtrada y posteriormente se le aplica una función no lineal para calcular la señal de salida R(t). Este es el modelo usado en los algoritmos de redes neuronales, ya que permite una computación muy eficiente con buenos resultados, al no ser necesario el estudio de cómo se comportan las señales dentro de la neurona, ni los cambios bioquímicos que suceden.

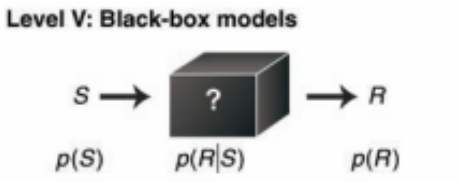

Figura 2.5: Modelos de caja negra **Fuente:** [4]

Por último, tenemos el modelo más simple de todos en la figura 2.5, el cual es simplemente una ”caja negra”, es decir, no conocemos los detalles de lo que ocurre dentro. Para modelizarla, se utiliza un modelo probabilístico, en el que conocemos las probabilidades de que, dada una entrada S, obtener una salida R (P(R|S)), donde la salida R viene dada de forma aleatoria con una probabilidad dependiente de este valor.

2.2. Enfoque matemático

Las redes neuronales se basan en un problema matemático: tenemos una función de error desconocida, de la que tenemos valores de entrada y salida, y de la cual queremos encontrar su mínimo global para poder obtener los mejores resultados posibles con esa función.

El modelo usado en las redes neuronales es el modelo de cascada [7], basado en el modelo presentado en la figura 2.5. En este caso, tenemos una función deseada $f : R^N \rightarrow R$ de la que desconocemos su forma, pero podemos calcular el error de un resultado dando una entrada. Contamos con un número de pares de ejemplos (x^μ, y^μ) donde $y^\mu = f(x^\mu)$ y $\mu = 1, ..., p$ siendo p el número de ejemplos.

La red está formada por varias neuronas, unidas en forma de capas, unidas también entre si. La topología de la red es la que define la función de salida de la red. Cada neurona tiene un número de variables constantes llamadas pesos. Es modificando estos pesos que la red consigue aprender. Es decir, intentamos encontrar los valores de la función definida por la red que minimice el error resultante.

La red consiste de N capas de neuronas, en el que la salida de una capa depende de cada una de las neuronas α, que, para una entrada x, la salida viene dada por la función:

$$\overline{V}(x) = \sum_{\alpha} \omega_\alpha V^\alpha(x) \tag{2.1}$$

Aquí, ω_α es el peso de cada entrada en la salida. Los valores ω_α son los valores que se van modificando a lo largo del tiempo mientras la red va aprendiendo.

El valor que intentamos minimizar es el error cuadrático medio de la salida de la red $\overline{V}(x)$ con respecto a la salida real de la función $f(x)$. El algoritmo más usado es el algoritmo "Adaptive Moment Estimation" o ADAM [11], el cual está basado en optimización estocástica basada en gradientes. El funcionamiento está explicado en el apéndice I. Otro algoritmo empezado a usar recientemente es "Adaptive Learning Rates with Sublinear Memory Cost" o ADAFACTOR [16], el cual intenta solventar los problemas de memoria que tiene ADAM en redes neuronales grandes.

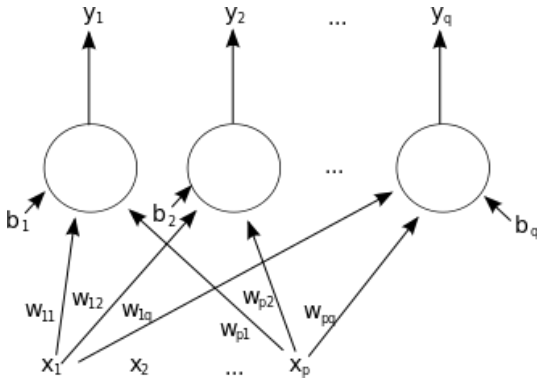

Figura 2.6: Red neuronal **Fuente:** Wikipedia

Capítulo 3
Recurrent Neural Networks

Tras haber explicado las redes neuronales básicas, tenemos que introducir las redes neuronales recurrentes. Su formulación surge ante un problema que tienen las redes neuronales simples, y es la incapacidad de que una entrada afecte a la siguiente salida, es decir, la misma entrada dará siempre la misma salida. El problema se puede ver al intentar aplicar redes neuronales a algunas tareas, por ejemplo el tratamiento del texto, en el que cada palabra depende de las anteriores y posteriores. Para solucionar esto, se crean neuronas con variables internas que permiten almacenar variables y que se modifican cada vez que se calcula una salida. Una de las topografías de red más común recurrente es la 'Long Short-Term Memory', o LSTM [5].

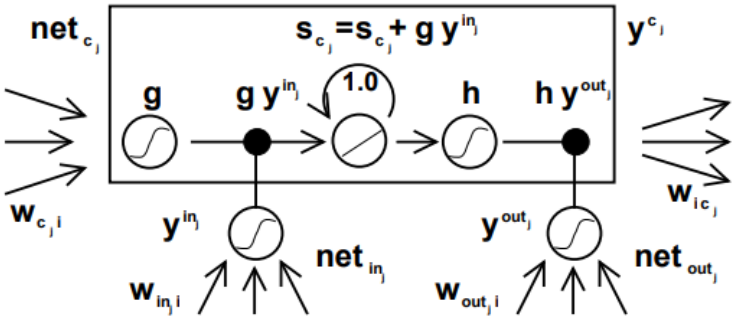

Figura 3.1: LSTM **Fuente:** [5]

Las redes LSTM están formadas por unos bloques o çells"que permiten almacenar datos de una iteración a la siguiente, sin los errores que surgen al hacerlo con redes tradicionales (anexo II).

Estas celdas calculan su salida $y^{c_j}(t)$, siendo j el índice de la celda, dada una entrada $y^u(t-1)$. Primero, definimos las variables $net_X(t)$ como:

$$\begin{aligned} net_{c_j}(t) &= \sum_u w_{c_j u} y^u(t-1), \\ net_{in_j}(t) &= \sum_u w_{in_j u} y^u(t-1), \\ net_{out_j}(t) &= \sum_u w_{out c_j u} y^u(t-1) \end{aligned} \tag{3.1}$$

Estas variables son pasadas por una función sigmoide. Nos referimos a la función sigmoide como f en caso de ser para las puertas de entrada y salida (in/out), como g para calcular la "memoria", y como h

para calcular la salida. De esta forma, definimos:

$$y^{in_j}(t) = f_{in_j}(net_{in_j}(t))y^{out_j}(t) = f_{out_j}(net_{out_j}(t)) \tag{3.2}$$

Con esto, ya podemos construir la celda. Lo primero, la entrada $net_{c_j}(t)$ se hace pasar por la función g y se multiplica por $y^{in_j}(t)$. Este valor es el que se almacena dentro de la celda. Si llamamos $s_cj(t)$ al valor que tiene almacenado en un tiempo t, este toma el valor:

$$s_{cj}(t) = \begin{cases} s_{cj}(t-1) + y^{in_j}(t)g(net_{c_j}(t)) & \text{if } t \neq 0 \\ 0 & \text{if } t = 0 \end{cases} \tag{3.3}$$

De esta forma, en cada paso de tiempo, el valor almacenado dentro de la celda se actualiza, teniendo en cuenta la entrada y los pesos de entrada, tanto w_{c_j} como w_{in_j}. Este valor se pasa por la función h, y la salida de la celda es el valor resultante multiplicado por $y^{out_j}(t)$:

$$y^{c_j} = y^{out_j}(t)h(s_{cj}(t)) \tag{3.4}$$

A parte de los pesos de entrada (w_{c_j}), que se comportan igual que los de una neurona normal, a las LSTM se les añade pesos de puerta de entrada (w_{in_j}) y de puerta de salida (w_{out_j}), que controlan cómo varía el valor almacenado y la salida, para evitar los errores que surgían al intentar almacenar valores en redes tradicionales (anexo II).

Uno de los principales problemas que se solucionan con esta arquitectura es el desvanecimiento de gradiente. Los algoritmos de entrenamiento modifican los pesos de una neurona dependiendo de la derivada parcial de la función de error con respecto a cada peso. Esto puede llevar al estancamiento cerca de un mínimo local, o a la parada del entrenamiento si el gradiente es muy pequeño.

Este tipo de redes supusieron una mejora con respecto a las redes tradicionales en un amplio número de casos, por ejemplo en el reconocimiento de voz, o el reconocimiento de texto manuscrito.

Sin embargo, debido a la aparición de más pesos por cada celda, necesitan de tiempos mayores para poder ser entrenadas, lo que hace que se intenten buscar alternativas, como los transformers que veremos en el Capítulo 5.

Capítulo 4
Encoders, Decoders and Attention Mechanism

En este capítulo, estudiaremos módulos creados específicamente para la tarea de tratamiento de texto, usando como ejemplo el modelo "Seq2Seq", desarrollado por Google [17].

Este modelo se basa en la utilización de un codificador o encoder, un decodificador o decoder, y un mecanismo de atención intermedio, el cual sirve de ”comunicación” entre ambos.

La estructura simplificada del modelo sería:

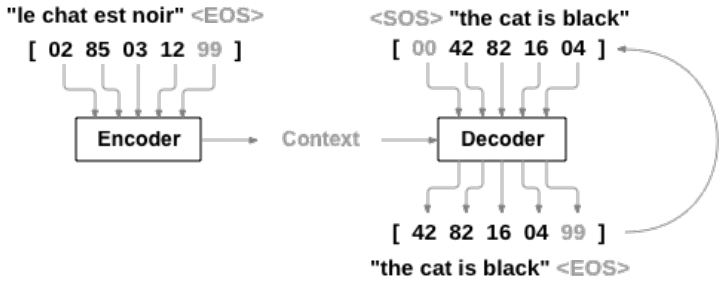

Figura 4.1: Seq2Seq **Fuente:** PyTorch Website [12]

En este caso, se está usando como traductor de textos, pero se puede utilizar también para otras tareas, tales como modelos conversacionales o para resumir textos.

El primer paso es cómo entiende la red los datos. Una red neuronal trabaja con tensores, mientras que el texto son cadenas de bytes (en formato ASCII) que pueden representar una frase, un diálogo, un libro, etc. Para convertir el texto a tensor, utilizamos un tokenizador.

Un tokenizador permite convertir una palabra (o parte de ella) en un *token*, al cual se le asigna un número único. Así, por ejemplo, la palabra ”árbol” se convierte en el token <árbol>, mientras que la palabra ”jugaba” se convierte en dos tokens, <jug>y <_aba>, cada uno con un id distinto.

Una vez podemos convertir cada palabra en un número, podemos convertir el texto en un tensor. Este tensor, de dimensión uno, tiene tantos números como *tokens* haya en nuestro diccionario (más alguno auxiliar), con todos los valores inicializados a 0 menos el valor que corresponde al id del *token*, que se inicializa con un 1.

También es necesario declarar *tokens* especiales, como <EOS>, que indica el final de una frase (End Of Sentence), o <SOS>, usado para indicar el comienzo de la frase de salida (Start Of Sentence).

Estos tokens son pasados uno a uno al *encoder*, el cual consiste de una red basada en redes recurrentes, tales como la LSTM explicada con anterioridad, también llamada GRU (*Gate Recurrent Unit*). La función del *encoder* es la de crear una representación intermedia de la frase completa, ya que al estar formado por celdas recurrentes es capaz de ”recordar” las palabras que ya se han introducido, y no sólo las actuales.

Este paso intermedio es pasado al *decoder*. El *decoder* nos da, dado un estado y un token, los siguientes posibles tokens con sus probabilidades. La forma de elegir entre los tokens se explica en el Anexo III.

Una vez elegido el siguiente token, este se almacena y se pasa como parámetro al *decoder*, que nos dará el siguiente. Este paso se repite hasta que el *decoder* nos devuelve el token de fin de frase <EOS>. El primer token pasado al *decoder* debe ser el de inicio de frase <SOS>.

Posteriormente, estos tokens se vuelven a convertir en palabras usando otro diccionario en caso de distintos idiomas, o el mismo si no se trata de un traductor.

En algunos casos, se introduce un bloque entre el *encoder* y el *decoder*, llamado *attention mechanism*, o mecanismo de atención. Este mecanismo consiste de otra red neuronal, cuyo objetivo es controlar los pesos que tienen cada palabra del *encoder* en el *decoder*, haciendo que no todas las palabras valgan lo mismo dependiendo del contexto en el que se encuentren. Esto hace que la salida del *encoder* ya no necesite ser de un tamaño fijo, ya que el mecanismo de atención se encargará de elegir las partes que necesita el *decoder* para realizar su función [1].

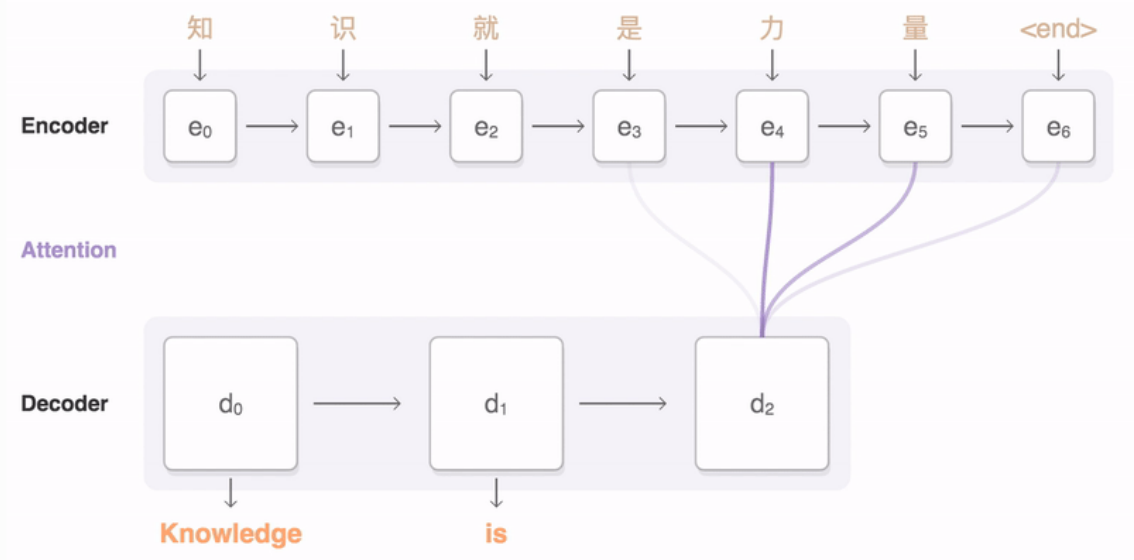

Figura 4.2: Seq2Seq Attention **Fuente:** Google Seq2Seq Website [17]

Los mecanismos de atención son muy importantes, ya que veremos a continuación que pueden usarse en redes neuronales no recurrentes, llegando a lo que se conoce como *Transformers*.

Capítulo 5

Transformers

Uno de los principales problemas de los primeros modelos de tratamiento de texto (por ejemplo, Seq2Seq [17]) es que dependían de neuronas con artquitectura LSTM, cuyo algoritmo de aprendizaje es menos óptimo que el de las redes no recurrentes, lo que aumenta el tiempo necesario. Por esto, en [19], se propone un nuevo modelo que utiliza redes neuronales no recurrentes para el tratamiento de texto. Este modelo recibe el nombre de Transformer.

La estructura es parecida al modelo Seq2Seq, seguimos teniendo un Encoder y un Decoder. Sin embargo, cada uno cuenta con una capa de mecanismo de atención, lo que nos permite no tener que usar LSTM. Otra diferencia es que, al no almacenar ningún dato de una iteración a la siguiente, es necesario introducir al encoder la frase entera. Posteriormente veremos cómo se realiza, ya que aunque cambia con respecto al Seq2Seq, sigue usando el mismo sistema de tokens.

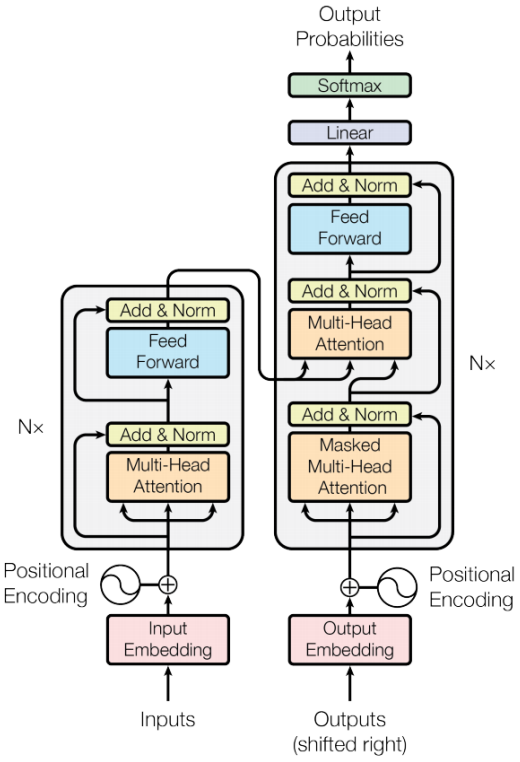

Figura 5.1: Transformer **Fuente:** [19]

Como se puede ver en la Figura 5.1, seguimos teniendo un encoder (parte izquierda) y un decoder (parte derecha). En este modelo puede haber más de una capa, tanto de encoder como de decoder, siendo Nx el nombre de la capa, con x el número de la capa. El modelo desarrollado en [19] utiliza 6 capas de encoder y 6 capas de decoder, mientras que otros modelos conocidos usan 8, 10, ó 12 capas.

El encoder consiste de dos sub-capas. La primera consiste en un mecanismo de atención multi-capa, o "Multi-Head Attention Mechanism". Para explicarlo, primero definimos la atención de una sola capa como:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{5.1}$$

Donde Q, K y V son matrices que definen las Q(ueries) o consultas, y K(eys) y V(alues) son los pares clave-valor. Aquí, se calcula el producto escalar de las consultas con las claves, se normaliza dividiéndolo para $\sqrt{d_k}$, y se le aplica una función softmax (función exponencial normalizada). El valor resultante el es peso para el valor V.

Si no normalizasemos los valores dividiendo por $\sqrt{d_k}$, este modelo de atención sería igual que el modelo de atención por producto escalar. Sin embargo, este modelo no tenía en cuenta la adición de varias capas, lo que hace que el producto escalar aumente con el número de dimensiones d_k, por lo que es necesario dividir por este valor para normalizar los resultados, ya que en caso contrario, la función softmax daría un valor muy extremal, donde el gradiente es casi nulo, ralentizando el proceso de aprendizaje.

Si colocamos varias capas conjuntas, podemos obtener lo llamado Multi-Head Attention. Esto nos permite extraer información de diferentes subespacios de representación en diferentes posiciones, algo que es imposible con una sola capa.

Así, definimos:

$$Multihead(Q, K, V) = Concat(head_1, head_2, ..., head_h)W^O \tag{5.2}$$

Donde

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{5.3}$$

Siendo W las matrices de los pesos que se modifican durante el entrenamiento. En la Figura 5.2 podemos ver una representación gráfica de estas capas.

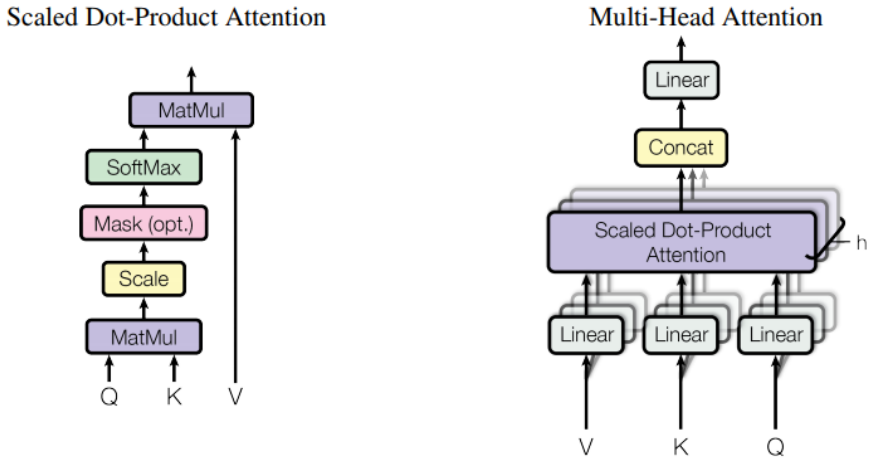

Figura 5.2: Attention mechanisms **Fuente:** [19]

La segunda sub-capa que encontramos dentro del encoder recibe el nombre de "Feed-Forward Network", la cual es una red neuronal simple completamente conectada, cuya salida para una entrada x es:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{5.4}$$

Por otro lado, el decoder se compone de los mismos componentes, a los que se le añade otro mecanismo de multi-atención, que toma como entrada la salida del decoder. La sub-capa de atención de entrada elimina los valores de las palabras que aún no han pasado por el modelo, para así evitar el flujo de información hacia la izquierda. Para esto, todos los valores que corresponden a conexiones ilegales toman el valor de $-\infty$.

Por último, necesitamos una forma de solventar la falta de información sobre la posición de cada token. Cuando usábamos redes recurrentes como la LSTM, al pasar un token cada vez, la información de su posición quedaba guardada. Sin embargo, al eliminarlas y pasar ahora todos los tokens en una única matriz, esta información se pierde. Para solventarlo, se utiliza el "Positional Encoding", en el que calculamos un valor que depende de la posición de cada token, y este valor se le añade a los tokens de entrada. En [19] proponen como función:

$$PE_{pos,2i} = sin(pos/10000^{2i/d_{model}}) \tag{5.5}$$

$$PE_{pos,2i+1} = cos(pos/10000^{2i/d_{model}}) \tag{5.6}$$

Siendo pos la posición del token e i la dimensión. Estos valores tienen cierta periodicidad para evitar que las redes utilicen sólo los valores absolutos de posición, y en su lugar utilicen los valores relativos.

Con estos cambios con respecto al Seq2Seq, este modelo es capaz de conseguir mejores resultados en distintas tareas, tales como la traducción o la categorización de textos. Además, como ya hemos comentado, al no depender de redes recurrentes no precisa de tiempos elevados de entrenamiento, consiguiendo alcanzar resultados anteriores en mucho menos tiempo.

Capítulo 6
ELMO, BERT and GPT-2

Tras haber visto las bases de los distintos modelos, pasamos a estudiar distintas implementaciones de estos, empezando por ELMO [13], una red basada en LSTM, GPT [14], BERT [2] y GPT-2 [15], que son redes basadas en transformers. Vamos a estudiar las diferencias que aparecen, y qué resultados obtenemos con estos modelos.

6.1. ELMO

ELMO (*Embeddings from Language Models* [13]) utiliza modelos de lenguaje bidireccionales (biLM). Para entender el modelo biLM, necesitamos explicar primero el modelo "backward LM":

Dada una secuencia de N tokens $(t_1, t_2, ..., t_N)$, el modelo trata de predecir el token en la posición k t_k dados todos los tokens anteriores $(t_1, t_2, ..., t_{k-1})$, siendo la probabilidad:

$$p(t_1, t_1, ..., t_N) = \prod_{k=1}^{N} p(t_k | t_1, t_2, ..., t_{k-1}) \tag{6.1}$$

Para esto, se calcula una representación de token x_k^{LM} y se pasa por L capas de redes LSTM. En cada posición k, cada capa tiene una salida $\vec{\mathbf{h}}_{k,j}^{LM}$, con j = 1, ..., L. La salida de la última capa, $\vec{\mathbf{h}}_{k,L}^{LM}$ es la usada para calcular el siguiente token t_{k+1}, pasando el resultado por una capa softmax.

Esta misma capa puede utilizarse como "forward LM", la cual nos intenta obtener el token actual dados los tokens futuros:

$$p(t_1, t_1, ..., t_N) = \prod_{k=1}^{N} p(t_k | t_{k+1}, t_{k+2}, ..., t_N) \tag{6.2}$$

Siendo el resultado de cada capa $\overleftarrow{\mathbf{h}}_{k,j}^{LM}$.

Una red biLM combina ambas, de tal forma que se intenta maximizar el resultado de las dos direcciones, es decir:

$$\sum_{k=1}^{N} (log(p(t_k | t_1, ..., t_{k-1}; \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s)) + log(p(t_k | t_{k+1}, ..., t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s))) \tag{6.3}$$

Siendo Θ_x la representación en forma de tokens, y Θ_s la salida de la capa Softmax.

ELMO está construido con estas capas biLM. Para cada token t_k, una capa L calcula 2L + 1 representaciones

$$R_k = \{x_k^{LM}, \vec{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} | j = 1, ..., L\} = \{\mathbf{h}_{k,j}^{LM} | j = 1, ..., L\} \tag{6.4}$$

Donde $\mathbf{h}_{k,0}^{LM}$ es la capa de tokens, y $\mathbf{h}_{k,j}^{LM} = [\overrightarrow{\mathbf{h}}_{k,j}^{LM}, h\overleftarrow{\mathbf{h}}_{k,j}^{LM}]$ para cada capa biLM.

Como resultado, se da un sólo vector

$$\mathbf{ELMo}_k = E(R_k; \Theta_e) \tag{6.5}$$

En el caso más simple, se elige la capa superior $E(R_k) = \mathbf{h}_{k,L}^{LM}$, aunque también se puede hacer un cálculo con pesos:

$$\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,L}^{LM} \tag{6.6}$$

Donde s^{task} son los pesos normalizados y γ^{task} es un parámetro escalar que nos permite escalar el resultado, lo cual ayuda en el proceso de aprendizaje.

6.2. GPT

GPT (*Generative Pre-trained Transformer* [14]) introduce la técnica del pre-entrenamiento. En este modelo, se utiliza un entrenamiento no regulado para entrenarlo, y después se le añade una capa de salida lineal, la cual se entrena para un proceso en concreto.

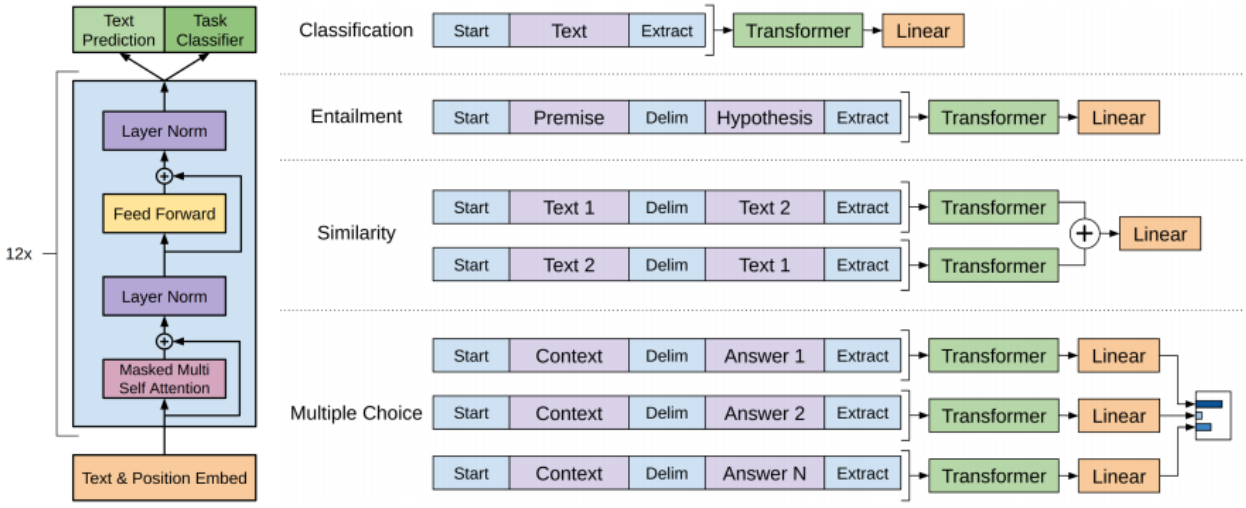

Figura 6.1: GPT **Fuente:** [14]

En la Figura 6.1 tenemos (izquierda) el modelo usado, un transformer como hemos visto en el Capítulo 5, al cual se le añade una capa lineal (derecha) para resolver diferentes problemas, desde clasificación de textos, hasta la elección múltiple.

El pre-entrenamiento se realiza sobre un corpus de N tokens $(u_1, ..., u_N)$, y se intenta maximizar

$$\sum_i log(p(u_i | u_{i-k}, ..., u_{i-1}; \Theta)) \tag{6.7}$$

Siendo k el tamaño del contexto (no todo el corpus está relacionado entre si, si no en "cajas" de tamaño k, que puede ser variable). El modelo usado da como salida:

$$h_0 = UW_e + W_p \tag{6.8}$$

$$h_l = transformer(h_{l-1}) \forall i \in [1, n] \tag{6.9}$$

$$p(u) = softmax(h_n W_e^T) \tag{6.10}$$

Donde $U = (u_{-k}, ..., u_{-1})$, n es el número de capas, W_e es la matriz de "embedding", y W_p es la matriz de valores de posición.

Una vez el modelo ya está pre-entrenado, se puede especializar en una tarea. Para esto, suponemos una secuencia de tokens de entrada $(x^1, ..., x^m)$ junto con una etiqueta y. Esta entrada se pasa por las capas de transformers, dando al final una salida $\mathbf{h}_l^m$, la cual se hace pasar por una capa lineal con los parámetros W_y tratando de predecir y:

$$p(y|x^1, ..., x^m) = softmax(\mathbf{h}_l^m W_y) \tag{6.11}$$

Lo cual nos da una función a maximizar:

$$\sum_{(x,y)} log(p(y|x^1, ..., x^m)) \tag{6.12}$$

6.3. BERT

BERT (*Bidirectional Encoder Representations From Transformers* [2]) utiliza también un método de pre-entrenamiento. Pero, a diferencia de GPT, no utiliza un "Masked Multi-Layer Attention Mechanism", lo que hace que el aprendizaje no vaya de izquierda a derecha, convirtiéndose en un transformer bidireccional.

Otra diferencia importante es que BERT premite introducir dos sentencias de token en un mismo input. Para esto, a parte de sumarse los tokens con sus posiciones, se les añade además un valor que indica en qué secuencia se encuentran (si sólo hay una secuencia, son todos iguales).

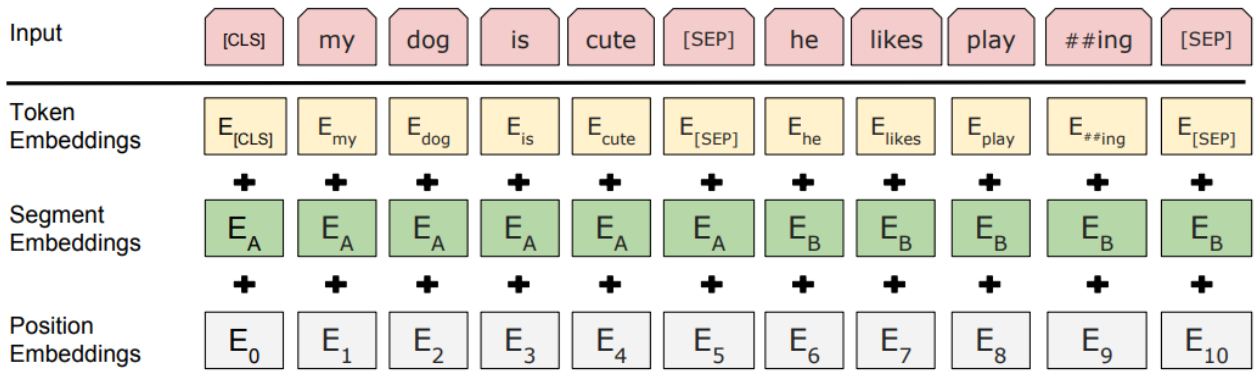

Figura 6.2: Token Representation BERT **Fuente:** [2]

Al ser bidireccional, el entrenamiento no puede ser como en los anteriores ejemplos, ya que las capas serían capaces de "ver" la palabra que están intentando adivinar. Para esto, se realizan dos pre-entrenamientos especiales.

En el primero, se elimina una palabra de una sentencia, y se intenta adivinar cuál es esa palabra. Para esto, se elige una palabra al azar, y dependiendo de un número aleatorio se realiza una acción u otra. Por ejemplo, si tenemos la frase "Mi perro es pequeño" y se elige la palabra "pequeño", puede ocurrir:

1. En un 80 % de las ocasiones, la palabra se cambia por el token [mask]: “Mi perro es [mask]“
2. En un 10 % de las ocasiones, la palabra se cambia por otra aleatoria: “Mi perro es árbol“
3. En un 10 % de las ocasiones, la frase se deja como está: “Mi perro es pequeño“

En el segundo, se siguen utilizando frases con palabras ocultas (usando [mask]), pero se entrena al modelo para adivinar si una frase es la siguiente a una dada. Para esto, se pasan dos frases, y el modelo devuelve una etiqueta “IsNext“ en caso de ser la siguiente frase, o una etiqueta “NotNext“ en caso de no serlo. Este modelo se entrena con una distribución 50/50 de frases correctas y frases incorrectas.

Al igual que en el modelo GPT, para hacer el entrenamiento final, se pasa la salida del transformer a una red neuronal lineal, que es la encargada de dar un resultado para cada tarea que se necesite.

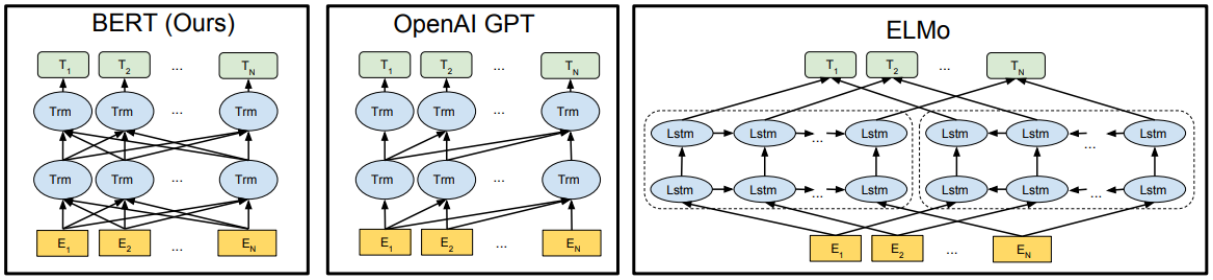

Figura 6.3: Diferencias ELMO, GPT, BERT **Fuente:** [2]

6.4. GPT-2

GPT-2 [15] utiliza un modelo similar a GPT, pero varía el cómo construye las tareas. Todos los modelos hasta ahora, se han basado en calcular un token s_n dada una secuencia $(s_1, ..., s_{n-1})$, cuya probabilidad se puede escribir como

$$p(x) = \prod_{i=1}^{n} p(s_n|s_1, ..., s_{n-1}) \tag{6.13}$$

Esto quiere decir, que para una tarea cualquiera, se intentaba calcular la distribución de probabilidad p(salida|entrada). GPT-2 utiliza un modelo en el que la tarea también se pasa como parámetro, es decir:

$$p(salida|entrada, tarea) \tag{6.14}$$

Para esto, construye frases específicas que indiquen la tarea a realizar. Por ejemplo, para traducir un texto del inglés al español, la tarea sería “traduce al español, texto en inglés, texto en español“, mientras que para responder a una pregunta sería “contesta a la pregunta, documento con datos, pregunta, respuesta“.

Además, mueve la función de normalización a la entrada de cada capa, y añade otra más después de la sub-capa de Self-Attention en la capa final.

Es un modelo que no cambia mucho a nivel de capas de redes, sin embargo, con el cambio en la representación de las entradas, consigue mejorar los resultados en algunos tipos de tareas, siendo además un modelo que permite tener varias soluciones en la misma red.

Capítulo 7
GPT-2 en español

En esta sección, vamos a desarrollar un modelo GPT-2 en español. Va a ser un modelo genérico, es decir, sólo realizaremos la parte del pre-entrenamiento, quedando un modelo entrenado listo para usarse en diferentes tareas.

Para aplicarlo, bastaría con añadir una (o varias) capas neuronales a la salida del transformer. Estas han de ser entrenadas para la tarea específica, pero el propósito de tener un núcleo ya entrenado en un idioma, es reducir drásticamente el tiempo de entrenamiento de esta segunda red, pasando de tener que entrenar semanas o meses para cada modelo, a entrenar una vez el núcleo durante un mes y todos los submodelos unos pocos días.

7.1. Dataset

Lo primero que necesitamos es un conjunto de datos lo suficientemente grande para poder entrenar nuestro modelo. El modelo usado por [15] utilizaba la base de datos de Reddit, un portal de internet con un gran número de usuarios y mensajes. Al no tener un equivalente del mismo tamaño en español, realizamos el pre-entrenamiento con la base de datos de Wikipedia en español.

Una vez descargado, esta base de datos consta de miles de carpetas y archivos, cada uno con una entrada de la web. Realizamos un primer proceso de limpieza, en el que se comprueba la dimensión de cada archivo, quitando los archivos más pequeños y moviendo el resto a una sola carpeta, para manipularlos de forma más sencilla.

Tras esto, extraemos de cada archivo el texto base, y lo copiamos todo a un sólo fichero, quedando cada párrafo en una línea del fichero. Con esto, tenemos un número de líneas $n = 1939032$. Este archivo es pasado por un tokenizador para convertir cada palabra en un número (llamado id), y guardado como tal, para evitar realizar la misma operación siempre que se quiera iniciar el entrenamiento.

Posteriormente, dentro del algoritmo de entrenamiento, uniremos todos estos párrafos usando un token especial, quedando del estilo:

$$... < text1 > [EOS] < text2 > ... \tag{7.1}$$

Este texto es luego cortado en trozos de un tamaño fijo:

$$d_1 = \{t_0, t_1, ..., t_k\} \tag{7.2}$$

$$d_2 = \{t_s, t_{s+1}, ..., t_{s+k}\} \tag{7.3}$$

$$d_3 = \{t_{2s}, t_{2s+1}, ..., t_{2s+k}\} \tag{7.4}$$

$$\vdots$$

Siendo k el tamaño del tensor de entrenamiento ($k = 1024$ en nuestro caso), y s un número de "salto" para controlar el tamaño del dataset, el cual tiene un valor de $s = 100$, dando un total de $n = 1456649$ frases para el entrenamiento, lo cual no es tan grande como el entrenamiento realizado en el *paper* que estamos estudiando [15], pero es suficiente para construir un pequeño modelo.

También construimos otro conjunto de datos más pequeño para poder probar la red en tiempos de entrenamiento más corto. Este dataset lo construimos a partir de la base de datos de la serie de televisión *friends*, el cual consta de un total de $n = 46446$ frases para el entrenamiento.

7.2. Modelo

El modelo lo dividimos en cuatro partes: El propio modelo de la red, un tokenizador, el código encargado del entrenamiento, y el código encargado de mostrar los resultados. Todos estos códigos están escritos en Python.

Estos códigos han sido escritos a partir de varias fuentes. Esto es debido a que existe un código para utilizar el modelo GPT-2 ya programado, pero este no permite el entrenamiento del núcleo, solamente de las capas añadidas a continuación del modelo. Por eso mismo, y usando este código como guía, ha sido necesario escribir el código de entrenamiento por completo.

Para el tokenizador utilizamos SentencePiece [8], un tokenizador desarrollado por Google que permite ser entrenado en cualquier idioma en apenas 10 minutos. Utilizamos el mismo dataset para entrenarlo, dando lugar a un tokenizador en español con el mismo vocabulario que el dataset.

SentencePiece separa las palabras en sub-palabras (*Byte Tokenizer*), y posteriormente les asigna un número de identificación único. Esta separación es sub-palabras se realiza para reducir el tamaño del vocabulario, sobre todo en idiomas con verbos conjugados, ya que permite guardar una única vez cada verbo, en lugar de una por cada persona y tiempo verbal (también permite reducir en sustantivos con sufijos y prefijos).

El modelo de la red se basa en una clase que tiene todos los métodos necesarios para calcular las salidas de la capa de transformer de forma sencilla. Tanto el código del modelo como de el entrenamiento y el muestreo de los resultados se encuentra en el anexo IV.

7.3. Resultados

Los entrenamientos se han realizado en los servidores del Instituto Tecnológico de Aragón, donde se contaba con la capacidad de memoria RAM necesaria (500GB) debido al gran tamaño del dataset, y a una gráfica especializada en entrenamiento para redes neuronales (NVIDIA Tesla V100).

La red consta de las mismas capas que el modelo general de GPT-2, formadas cada una de ellas por 1024 neuronas completamente conectadas entre si. El tamaño de la red en memoria es cercano a los 32GB.

El entrenamiento con el primer conjunto de datos (Wikipedia) se realizó durante un mes, debido al tiempo que tarda en entrenar una sola época. Se llama época a un recorrido completo por todos los datos de entrenamiento, es decir, cuando todos los datos de entrenamiento (divididos en *batches*) han pasado mínimo una vez por la red neuronal, se dice que se ha completado una *época*.

Tras entrenar 10 épocas con una duración de 21 días, observamos los siguientes resultados:

- El tiempo de entrenamiento es elevado, tardando varios días en completar una época.

- El error disminuye rápidamente al principio del entrenamiento, pero tiende a estancarse en un valor tras varias iteraciones. El error sigue disminuyendo, pero a un ritmo muy bajo.
- Tras el entrenamiento, se observan comportamientos en el resultado correctos (por ejemplo, poner los años entre paréntesis) pero el texto completo carece de significado.
- Para poder entrenar un modelo tan grande, haría falta mucho tiempo de entrenamiento, o más capacidad de cálculo.

Para obtener estos resultados, hemos suministrado al modelo un tensor vacío, al cual añadimos una palabra. El modelo predice la siguiente palabra, la cual se añade al tensor, y se vuelve a pasar al modelo. Este ciclo se repite hasta que se alcanza el tamaño máximo de palabras (1024 en nuestro caso), o el modelo devuelve el *token* de fin de frase. En este caso, las frases constaban de una media de 100 palabras.

A pesar de los pocos resultados de este entrenamiento, el modelo es capaz de relacionar algunas palabras entre si. En la tabla 7.1 mostramos palabras pasadas al modelo y palabras relacionadas que devolvía (hay que tener en cuenta que, junto a estas palabras, devuelve también un gran número de pronombres y preposiciones, lo que hace que la frase en conjunto no tenga sentido, pero se puede ver cómo ha formado ciertas relaciones).

Palabra introducida	**Palabras relacionadas**
Ciudad	Ciudad, ciudadano, gran
Castillo	Mayor, forma parte, ciudad, donde
Hierba	Paseos, lodo, arar

Tabla 7.1: Resultados dataset Wikipedia

Como el problema de la elevada duración del proceso de entrenamiento se empezó a ver a tiempo, probamos a crear un segundo dataset más pequeño para comprobar los resultados en un entorno más reducido.

Este segundo dataset está formado por las frases de los diálogos de la serie *friends*. El tratamiento de este texto se ha realizado igual que el de Wikipedia: se han unido todas las frases, y se ha recortado el texto resultante en ”trozos” de 1024 palabras.

Es importante recordar que no estamos intentando crear una red neuronal que sepa responder a frases, por lo que podemos perder la información del diálogo uniendo las frases sin preocuparnos, ya que el resultado debería ser el entendimiento del lenguaje.

Con este dataset hemos conseguido obtener frases con sentido. La frase entera devuelta por el modelo (unas 15 palabras) sigue sin tener coherencia, apareciendo palabras sin sentido o con poca conexión con el resto, pero se pueden observar pequeñas frases con sentido:

- *Oye, puedo?*
- *Si porque es mi*
- *Bueno, como no* (no ha sido capaz de *aprender* cuándo usar acentos)
- *No me puedo*
- *Todo lo que siento, pero vale*
- *Pero Ross(nombre de un personaje de la serie) puedo*

Como podemos ver, no son frases completas, pero ya se empiezan a encontrar relaciones entre palabras consecutivas, por lo que la red ha empezado a *aprender* la posición de algunas palabras dentro de la frase.

Tras entrenar este mismo modelo por 10 épocas más, notamos mejora en los resultados. La frase devuelta ahora está formada por varias frases con sentido, aunque no haya conexión entre ellas. Además, algunas respuestas empiezan a presentar los nombres de los personajes de la serie con mayor asiduidad, acompañados de un verbo, lo cual dota de mayor sentido a la frase.

Capítulo 8
Conclusiones y Trabajo Futuro

El procesamiento de texto es una tarea muy importante dentro de las nuevas tecnologías. Ha habido muchos avances en este área gracias al desarrollo de nuevos algoritmos de redes neuronales y al aumento de la capacidad de procesamiento de los ordenadores actuales.

Sin embargo, sigue siendo una tarea complicada de llevar a cabo con los recursos actuales. Con los recursos disponibles no podemos obtener los mismos resultados que obtienen las grandes compañías como OpenAI o Google, pero habiendo conseguido una poca coherencia en los resultados, sólo es necesario más tiempo para poder obtener mejores resultados con el modelo propuesto.

Esto es posible solucionarse de dos formas. La primera, con más iteraciones de entrenamiento, lo cual lleva a necesitar más tiempo y/o más potencia de cálculo. Lo máximo que se ha conseguido avanzar en este aspecto es el modelo más grande de GPT-2 [15], el cual genera textos coherentes que pueden ser pasados por humanos. Sin embargo, este modelo no llega a alcanzar resultados humanos en otros tipos de tareas (aunque ha conseguido superar a todos los modelos anteriores). Sin embargo, este modelo está entrenado en una base de datos mucho mayor (noticias enlazadas desde el foro *Reddit*), y disponían de varios núcleos de procesamiento especializados en el calculo tensorial (TPUs: *Tensor Processing Units*) trabajando en paralelo, o cual les permitía entrenar de forma más eficiente.

Otra solución es la creación de nuevos algoritmos que permitan un mejor resultado. En el tratamiento de imágenes, se están consiguiendo resultados mejores en tiempo de entrenamiento más cortos utilizando algoritmos de computación cuántica [10]. Estos resultados podrían aplicarse también al tratamiento de textos, pero el tema de la computación cuántica queda fuera de este trabajo.

Bibliografía

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[3] S. Dreiseitl and L. Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.

[4] A. V. Herz, T. Gollisch, C. K. Machens, and D. Jaeger. Modeling single-neuron dynamics and computations: a balance of detail and abstraction. *science*, 314(5796):80–85, 2006.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544, 1952.

[7] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. In *Advances in neural information processing systems*, pages 231–238, 1995.

[8] T. Kudo and J. Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium, Nov. 2018. Association for Computational Linguistics.

[9] I. Kulikov, A. H. Miller, K. Cho, and J. Weston. Importance of a search strategy in neural dialogue modelling. *arXiv preprint arXiv:1811.00907*, 2018.

[10] N. T. Nguyen and G. T. Kenyon. Image classification using quantum inference on the d-wave 2x. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–7. IEEE, 2018.

[11] D. P. Kingma and J. Lei Ba. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2015.

[12] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[13] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[14] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/languageunsupervised/language understanding paper. pdf*, 2018.

[15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1:8, 2019.

[16] N. Shazeer and M. Stern. Adafactor: Adaptive learning rates with sublinear memory cost. *arXiv preprint arXiv:1804.04235*, 2018.

[17] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[18] A. Topirceanu, G. Barina, and M. Udrescu. Musenet: Collaboration in the music artists industry. In *2014 European Network Intelligence Conference*, pages 89–94. IEEE, 2014.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[20] Y. Yang, L. Huang, and M. Ma. Breaking the beam search curse: A study of (re-) scoring methods and stopping criteria for neural machine translation. *arXiv preprint arXiv:1808.09582*, 2018.